30 | 预压测：如何基于小流量快速验证容量压测效果？

2021-12-25 高楼

《全链路压测实战30讲》 课程介绍 >

讲述：高楼

时长 08:26 大小 7.73M

你好，我是高楼。

上一节课呢，我们重点讲解了基准场景、容量场景、稳定性场景和异常场景这四类性能场景。通过对一个接口的基准场景模拟，我们提前发现了 Calico IPIP 模式单网卡软中断的问题，为后面容量场景的测试减少了一些麻烦。

这节课，我们就再进一步，把容量场景跑起来。我们先预压测一下，把场景中存在的小问题先找出来。

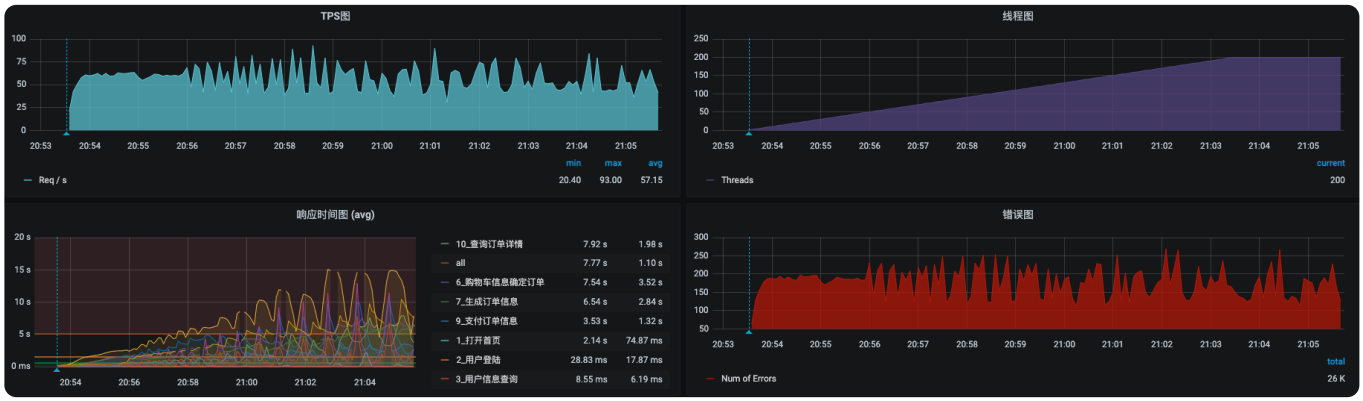
因为是预压测，所以一开始我们不用上太多的线程。先添加 20 个线程，运行一下看看结果就可以了。

说明一下，这里我换到 JMeter 上来运行，倒不是因为 GoReplay 不能满足压测需求，经过前面的改造，GoReplay 其实已经可以完成我们的压测需求了。只是我还是更习惯 JMeter 在功能上的丰富性，所以在预压测这一讲，我们还是用 JMeter 来执行。

第一阶段

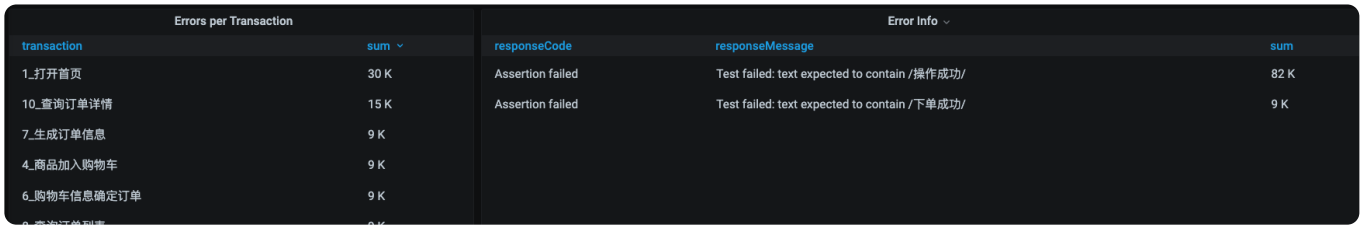
压力场景数据

怀着激动的心情，我们先来启动压力：



果然是不压不知道，一压心直跳呀。错误率很高不说，响应时间也很长。这就是我们要面对的问题了。

我们先来分析一下错误在哪。



Errors per Transaction

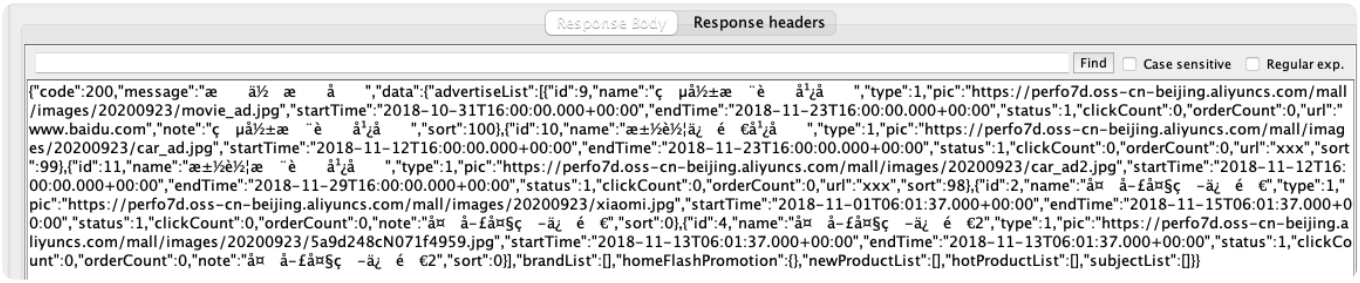
transaction	sum
1_打开首页	30 K
10_查询订单详情	15 K
7_生成订单信息	9 K
4_商品加入购物车	9 K
6_购物车信息确定订单	9 K

Error Info

responseCode	responseMessage	sum
Assertion failed	Test failed: text expected to contain /操作成功/	82 K
Assertion failed	Test failed: text expected to contain /下单成功/	9 K

从报错的信息来看，是断言判断不到设定的“操作成功”和“下单成功”字样。这是什么问题呢？

我们先把脚本拿到本地来跑一下试试。



我们发现，返回信息里的中文全是乱码。这样一来，断言自然也就失效了。为了解决乱码，我们打开 JMeter 目录中的 /bin/jmeter.properties，添加 UTF-8 的配置到文件中。

```
1 sampleresult.default.encoding=utf-8
```

复制代码

再次回放，脚本就可以正常显示了。

我们还是用不大的压力（20 个线程）再把压力跑起来，压力数据变成了下面这个样子：

看起来好像还不错。虽然也有几个错误，但相比刚才已经少很多了。剩下的这些错误应该就是真正的业务错误了。不过我们现在先不管它。先看看资源使用率的情况。

全局监控分析

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
s10	s10	1.49 week	15.00 GiB	8	0.47	8.36%	44.30%	41.25%	6.48 kB/s	1.55 MB/s	11	118	79.16 Mb/s	72.53 Mb/s
s11	s11	1.49 week	15.00 GiB	8	2.74	48.30%	62.22%	81.99%	152.64 kB/s	4.98 MB/s	22	264	131.60 Mb/s	76.80 Mb/s
s12	s12	3.95 day	15.00 GiB	8	7.86	87.91%	47.15%	45.21%	659.81 kB/s	6.13 MB/s	10	101	95.17 Mb/s	115.06 Mb/s
s5	s5	2.05 week	7.64 GiB	2	0.26	14.78%	32.42%	50.72%	0 B/s	99.79 kB/s	307	183	11.12 Mb/s	9.33 Mb/s
s6	s6	2.05 week	7.64 GiB	2	0.51	21.79%	25.34%	55.38%	117.03 B/s	1.64 MB/s	21	94	12.65 Mb/s	11.72 Mb/s
s7	s7	2.04 week	7.64 GiB	2	0.23	12.14%	75.14%	72.25%	117.03 B/s	250.01 kB/s	10	237	1.50 Mb/s	2.56 Mb/s
s8	s8	1.49 week	15.00 GiB	8	0.04	1.74%	47.80%	42.50%	0 B/s	15.80 kB/s	10	201	1.62 Mb/s	1.84 Mb/s
s9	s9	2.04 week	15.00 GiB	8	0.98	14.67%	65.11%	68.76%	53.64 kB/s	1.64 MB/s	11	466	79.04 Mb/s	99.56 Mb/s

通过这张全局监控截图，我们比较欣喜地看到，终于有一个机器的 CPU 已经标红了。在这种情况下，再分析就是要走我们的 RESAR 性能分析七步法了，保证手到擒来。

下面我们进到 CPU 高的 s12 服务器，执行 top。

```
top - 23:14:44 up 3 days, 22:55,  1 user,  load average: 14.43, 11.99, 8.53
Tasks: 193 total,   1 running, 192 sleeping,   0 stopped,   0 zombie
%Cpu0  : 76.4 us,  6.7 sy,  0.0 ni, 13.8 id,  0.3 wa,  0.0 hi,  2.7 si,  0.0 s
%Cpu1  : 71.4 us,  7.7 sy,  0.0 ni, 19.9 id,  0.3 wa,  0.0 hi,  0.7 si,  0.0 s
%Cpu2  : 76.5 us,  6.4 sy,  0.0 ni, 14.8 id,  1.0 wa,  0.0 hi,  1.3 si,  0.0 s
%Cpu3  : 79.9 us,  4.7 sy,  0.0 ni, 14.1 id,  0.7 wa,  0.0 hi,  0.7 si,  0.0 s
%Cpu4  : 85.6 us,  3.7 sy,  0.0 ni,  9.4 id,  0.3 wa,  0.0 hi,  1.0 si,  0.0 s
%Cpu5  : 81.4 us,  4.1 sy,  0.0 ni, 13.2 id,  1.0 wa,  0.0 hi,  0.3 si,  0.0 s
%Cpu6  : 87.6 us,  4.4 sy,  0.0 ni,  5.4 id,  0.7 wa,  0.0 hi,  2.0 si,  0.0 s
%Cpu7  : 85.2 us,  5.4 sy,  0.0 ni,  8.4 id,  0.7 wa,  0.0 hi,  0.3 si,  0.0 s
KiB Mem : 15731892 total,  1074660 free,  7042760 used,  7614472 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  8349352 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6515 27        20   0 3916212 325008  13304 S 547.0  2.1 715:56.39 /opt/rh/rh
 3868 root      20   0 9706460 804212  14332 S  61.3  5.1 113:43.58 java -Dcom
29812 1000      20   0 1942964 243780   9608 S  25.5  1.5   0:38.93 java -Xms5
25306 nfsnobo+  20   0 2599060 489856  51700 S   7.3  3.1  59:17.13 /bin/prome
 1189 root      20   0 1275732 108700  30124 S   6.6  0.7  92:33.30 /usr/bin/d
 1174 root      20   0 2271884 104784  36608 S   4.3  0.7 152:25.24 /usr/bin/k
22095 root      10 -10  200708  33180  15344 S   2.0  0.2   6:42.03 /usr/local
24216 1000      20   0 2068796 486356  13508 S   2.0  3.1   5:23.46 java -Xms5
 3850 root      20   0  109096   8148   2652 S   1.7  0.1   3:10.85 containerd
    1 root      20   0   44252   4672   2656 S   0.3  0.0   2:04.64 /usr/lib/s
  414 root      20   0  149560  77504  77136 S   0.3  0.5   2:19.20 /usr/lib/s
```

从 top 的数据来看，问题在于 mysql 消耗的 CPU 较高。

定向监控分析

在我的经验中，这种时候就可以直接到 innodb_trx 里面蒙一把了，里面大概率是能看到慢日志的。我们在查询 innodb_trx 表中的数据时要多查几次，如果同一事务 ID 持续了一段时间，那就是明显慢了。当然，如果你开启了慢日志，可以到慢日志里去查看。

trx_id	trx_state	trx_started	trx_requested_lock...	trx_wait_start...	trx_weight	trx_mysql_thread...	trx_query		trx_operation_st...	trx_tables_in_u...	trx_tables_lock...	trx_lock_stru...	trx_lock_memory_by...	trx
422074965061264	RUNNING	2021-12-01 23:19:36	NULL	NULL	0	10925	select	id, order_id, order_sn, product_...	fetching rows	1	0	0	1136	0
422074965060352	RUNNING	2021-12-01 23:19:36	NULL	NULL	0	10921	select	id, order_id, order_sn, product_...	fetching rows	1	0	0	1136	0
422074965057616	RUNNING	2021-12-01 23:19:36	NULL	NULL	0	10917	select	id, order_id, order_sn, product_...	NULL	1	0	0	1136	0
422074965056704	RUNNING	2021-12-01 23:19:36	NULL	NULL	0	10916	select	id, order_id, order_sn, product_...	NULL	1	0	0	1136	0
422074965052144	RUNNING	2021-12-01 23:19:36	NULL	NULL	0	10910	select	id, order_id, order_sn, product_...	fetching rows	1	0	0	1136	0
422074965050320	RUNNING	2021-12-01 23:19:36	NULL	NULL	0	10908	select	id, order_id, order_sn, product_...	fetching rows	1	0	0	1136	0
422074965047584	RUNNING	2021-12-01 23:19:36	NULL	NULL	0	10905	select	id, order_id, order_sn, product_...	fetching rows	1	0	0	1136	0
422074965045760	RUNNING	2021-12-01 23:19:36	NULL	NULL	0	10903	select	id, order_id, order_sn, product_...	fetching rows	1	0	0	1136	0

果不其然，我在这里确实抓到了可能比较慢的 SQL。我们取出它的执行计划看看。

id	select_type	table	partitions	type	possible_keys	key	key_l...	ref	rows	filtered	Extra
1	SIMPLE	o		const	PRIMARY	PRIMARY	8	const	1	100.00	
1	SIMPLE	ot		ALL					104303	100.00	Using where

因为 ot 表上是全表扫描，涉及到 10 万条数据，这数据太多了，又没走索引。我们执行语句，添加一个索引。

复制代码

```
1 alter table mall.oms_order_item add index index_order_id(order_id) ;
2 analyze table mall.oms_order_item;
```

再次查看执行计划看看区别：

id	select_type	table	partitions	type	possible_keys	key	key_l...	ref	rows	filtered	Extra
1	SIMPLE	o		const	PRIMARY	PRIMARY	8	const	1	100.00	
1	SIMPLE	ot		ref	index_order_id	index_order...	9	const	1	100.00	

这回，从 rows 那一列是不是看到了明显的变化？ot 表相关的 type 也变成了我们添加的索引。这种问题，应该说是非常常见了。

这时候我又刷新了几下 innodb_trx 表，又找了几个走全表扫描的 SQL。各自添加索引之后，我们再回去看一眼 TPS：

截图里这种情况就有点奇怪了呀。很显然在刚才改索引的时候，TPS 是上去过的，你可以看一下 23：24 分之前的那一段，TPS 曾经达到过 800 左右。但是中间却断了一会儿，这里应该是和我刚才执行的添加索引、分析表的 SQL 语句有关。我们再观察一会儿：

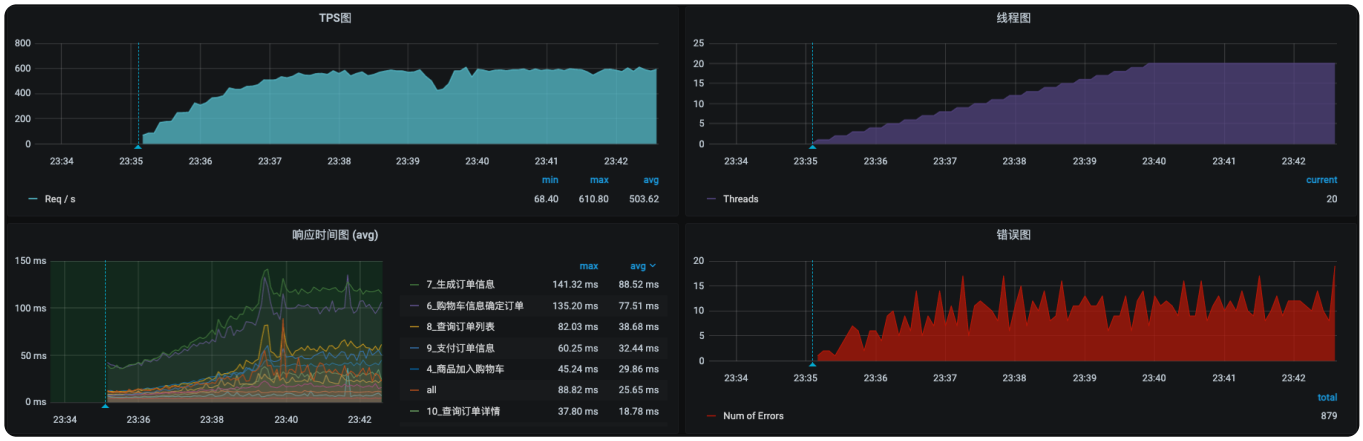
可以看到，报错之后压力并没有变化，但 TPS 是在逐渐恢复的。这是个好的方向。这时候我们再回来看看资源使用的情况：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
s10	s10	1.50 week	15.00 GiB	8	0.81	13.83%	44.21%	41.16%	696.59 kB/s	1.66 MB/s	10	131	92.55 Mb/s	87.83 Mb/s
s11	s11	1.50 week	15.00 GiB	8	4.60	19.16%	55.14%	61.68%	108.21 kB/s	3.23 MB/s	18	192	120.23 Mb/s	87.16 Mb/s
s12	s12	3.96 day	15.00 GiB	8	8.67	24.37%	49.84%	46.94%	369.81 kB/s	6.49 MB/s	11	184	107.87 Mb/s	98.71 Mb/s
s5	s5	2.05 week	7.64 GiB	2	0.71	17.84%	32.50%	50.72%	0 B/s	115.66 kB/s	301	177	13.30 Mb/s	10.68 Mb/s
s6	s6	2.05 week	7.64 GiB	2	6.41	95.09%	31.03%	58.04%	111.02 kB/s	4.16 MB/s	22	221	18.27 Mb/s	16.97 Mb/s
s7	s7	2.05 week	7.64 GiB	2	0.62	15.87%	75.29%	72.68%	78.02 B/s	492.65 kB/s	10	247	2.24 Mb/s	3.66 Mb/s
s8	s8	1.50 week	15.00 GiB	8	0.04	1.95%	48.00%	42.50%	0 B/s	14.63 kB/s	10	222	3.50 Mb/s	3.97 Mb/s
s9	s9	2.04 week	15.00 GiB	8	1.86	14.32%	65.21%	74.53%	55.43 kB/s	2.88 MB/s	11	469	41.21 Mb/s	54.47 Mb/s

还记得上面我们看到的是 s12 资源使用率多吗？现在已经转移到了 s6 身上了吧。这说明上一个问题已经解决，现在是个新问题了。

第二阶段

接下来我们再运行一段时间场景，看看还有啥问题。



这里还是有一些报错。我查了一下日志，发现是因为库存不足。

复制代码

```
2021-12-01 23:39:53.327 ERROR [mall-order,,,] 1 --- [io-8084-exec-18] o.a.c.c.

com.dunshan.mall.common.exception.ApiException: 库存不足，无法下单
  at com.dunshan.mall.common.exception.Asserts.fail(Asserts.java:11)
  at com.dunshan.mall.order.service.impl.PortalOrderServiceImpl.generateOrder(
  at com.dunshan.mall.order.service.impl.PortalOrderServiceImpl$$FastClassBySp
.....................
```

业务的问题比较好处理，加上库存就行了。

从上面这张图我们也可以看到，TPS 一直稳稳地在近 600 左右。

资源使用率方面呢，根据上面的全局监控，我们可以知道 s6 上的资源使用率过高。所以我们要登录到这台主机，看看是什么进程导致的。

复制代码

```
top - 23:45:54 up 14 days,  8:12,  2 users,  load average: 18.57, 16.12, 11.52
Tasks: 126 total,   2 running, 124 sleeping,   0 stopped,   0 zombie
%Cpu0  : 70.9 us, 13.6 sy,  0.0 ni,  2.3 id,  0.0 wa,  0.0 hi, 13.2 si,  0.0 s
%Cpu1  : 71.1 us, 16.5 sy,  0.0 ni,  8.9 id,  0.3 wa,  0.0 hi,  3.1 si,  0.0 s
KiB Mem :  8008964 total,   173104 free,  2266508 used,  5569352 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  5433884 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
31288 root      20   0 4624912 536084  12736 S 100.3  6.7  19:48.18 java -Dcom
 7931 root      20   0 4637452 697772  12728 S  60.5  8.7  90:39.72 java -Dcom
 1473 root      20   0  675600  94608  26768 S  16.6  1.2 540:03.39 /usr/bin/d
 1894 root      20   0 1729636  90460  35948 S   2.3  1.1 618:52.03 /usr/bin/k
21598 root      10 -10  197132  27972  15480 S   2.3  0.3  10:07.67 /usr/local
```

```
14 31271 root      20   0  107688   3380   2556 S   2.3  0.0   0:27.37 containerd
15  7913 root      20   0  109096   5620   2716 S   1.7  0.1   3:33.94 containerd
16     9 root      20   0       0      0      0 R   0.3  0.0  11:05.99 [rcu_sched
17   609 root      20   0  835660  26056   7852 S   0.3  0.3  58:34.33 CmsGoAgent
18  2927 1000      20   0  141496  29604  12708 S   0.3  0.4  13:10.48 /metrics-s
19 ..............
```

通过 top 中的 process table，我们看到上面有两个 Java 的进程，由于名字太长看不全，不知道是什么具体的服务。这种情况是有多个服务跑在了一个 2C 的 s6 上。接下来，我们查看一下 s6 上有什么 Pod：

复制代码

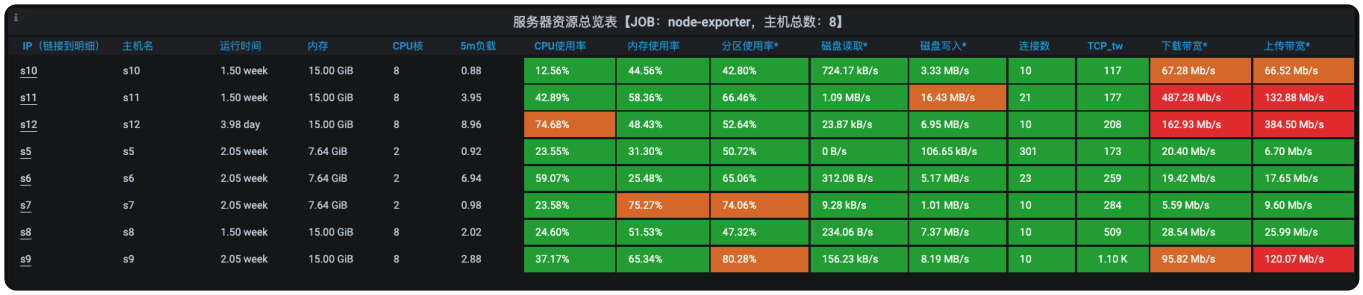
```
1 [root@s5 runMall]# kubectl get pods -n default -o wide | grep s6
2 kafka-1                                1/1     Running     48         13d
3 node-exporter-rqgrx                    1/1     Running     0          3d4h
4 svc-mall-cart-dd5db86d7-dcwdd          1/1     Running     0          3d23h
5 svc-mall-order-67cd58db5-ht5mb         1/1     Running     0          12m
6 [root@s5 runMall]#
```

果然，这里有两个服务，还都是重要的服务。对一个 2C 的虚拟机来说，能跑近 600 的 TPS，我觉得已经挺好的了，不能再指望更高的 TPS 了。为了让 TPS 再高一些，我把 Order 服务移到 s8 上去，因为 s8 感觉没用上。

把 Order 服务从 s6 移到 s8 之后，再执行一会场景，查看压力数据：

看起来不错哦，TPS 从 600 左右上升到了 960 左右。

接着我们再回来看一下全局监控数据：



服务器资源总览表【JOB：node-exporter，主机总数：8】

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
s10	s10	1.50 week	15.00 GiB	8	0.88	12.56%	44.56%	42.80%	724.17 kB/s	3.33 MB/s	10	117	67.28 Mb/s	66.52 Mb/s
s11	s11	1.50 week	15.00 GiB	8	3.95	42.89%	58.36%	66.46%	1.09 MB/s	16.43 MB/s	21	177	487.28 Mb/s	132.88 Mb/s
s12	s12	3.98 day	15.00 GiB	8	8.96	74.68%	48.43%	52.64%	23.87 kB/s	6.95 MB/s	10	208	162.93 Mb/s	384.50 Mb/s
s5	s5	2.05 week	7.64 GiB	2	0.92	23.55%	31.30%	50.72%	0 B/s	106.65 kB/s	301	173	20.40 Mb/s	6.70 Mb/s
s6	s6	2.05 week	7.64 GiB	2	6.94	59.07%	25.48%	65.06%	312.08 B/s	5.17 MB/s	23	259	19.42 Mb/s	17.65 Mb/s
s7	s7	2.05 week	7.64 GiB	2	0.98	23.58%	75.27%	74.06%	9.28 kB/s	1.01 MB/s	10	284	5.59 Mb/s	9.60 Mb/s
s8	s8	1.50 week	15.00 GiB	8	2.02	24.60%	51.53%	47.32%	234.06 B/s	7.37 MB/s	10	509	28.54 Mb/s	25.99 Mb/s
s9	s9	2.05 week	15.00 GiB	8	2.88	37.17%	65.34%	80.28%	156.23 kB/s	8.19 MB/s	10	1.10 K	95.82 Mb/s	120.07 Mb/s

资源使用率也比之前高了不少。

看起来，我们的系统已经要进入到正常的系统状态了，到这里，预压测也就可以结束了。下节课，我们就可以上更大一些的压力接着玩了。

总结

好了，这节课就讲到这里。刚才，我们对容量场景进行了预压测。

其实这个过程和正式的容量场景没有什么区别，只是压力线程少一点而已。预压测可以验证场景设置、参数化数据、网络环境、硬件环境等内容。它可以让我们在执行正式的场景的时候，不至于出现耗时的问题。像这节课讲的“索引”这种基础问题，就应该在预压测的时候就解决掉，不应该出现在大压力的场景之中。

在很多人的性能思路中，基准场景就是拿三五个压力线程运行一下，而预压测的过程则根本不存在。但是在我的 RESAR 性能工程的理念中，在真正模拟生产峰值的容量场景之前，是有必要加上预压测的部分的。预压测使用的是生产的业务模型，是混合的业务场景。预压测的存在可以给容量场景做好充分的铺垫。

课后题

学完这节课，请你思考两个问题：

1. 预压测是不是必须的呢？
2. 不做预压测会带来什么问题？

欢迎你在留言区与我交流讨论。当然了，你也可以把这节课分享给你身边的朋友，他们的一些想法或许会让你有更大的收获。我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 29 | 基准场景：一个案例，带你搞懂基准场景的关键要点

更多学习推荐

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。